CENG 3420 Computer Organization & Design

Lecture 06: Control Instruction

Bei Yu
CSE Department, CUHK
byu@cse.cuhk.edu.hk

(Latest update: January 27, 2022)

Spring 2022

Overview

1. Introduction
2. Control Instructions
3. Others
4. Summary

Overview

1 Introduction

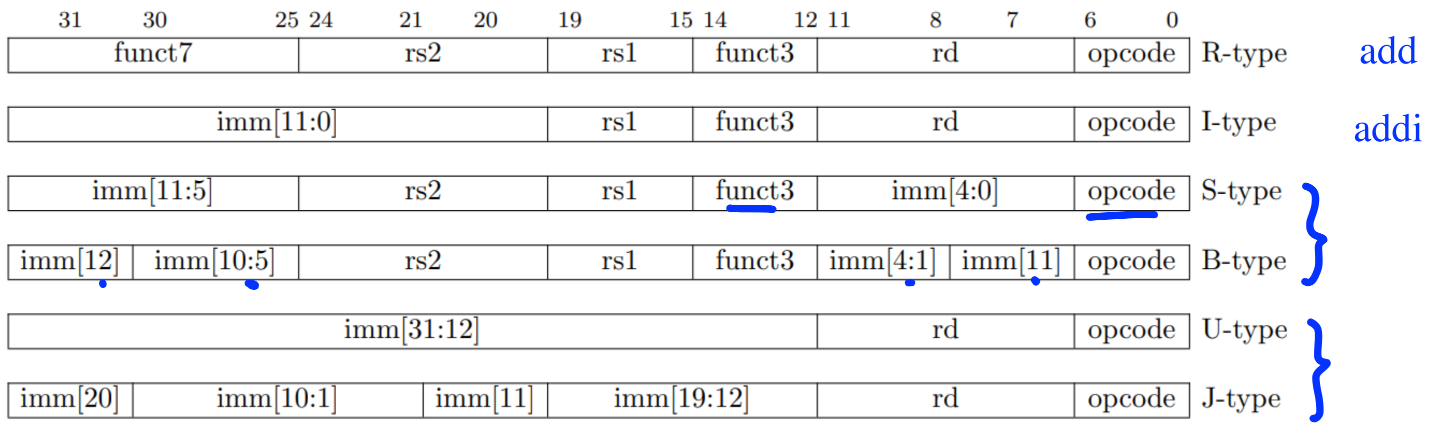
2 Control Instructions

3 Others

4 Summary

RISC-V Instruction Fields

RISC-V fields are given names to make them easier to refer to


31
30
25 24
21
20
19
15 14
12 11
8
7
6
0
funct7 | rs2 | rs1 | funct3 | rd | opcode — R-type
add
imm[11:0] | rs1 | funct3 | rd | opcode — I-type
addi
imm[11:5] | rs2 | rs1 | funct3 | imm[4:0] | opcode — S-type
imm[12] | imm[10:5] | rs2 | rs1 | funct3 | imm[4:1] | imm[11] | opcode — B-type
imm[31:12] | rd | opcode — U-type
imm[20] | imm[10:1] | imm[11] | imm[19:12] | rd | opcode — J-type

B-format: no imm[0]

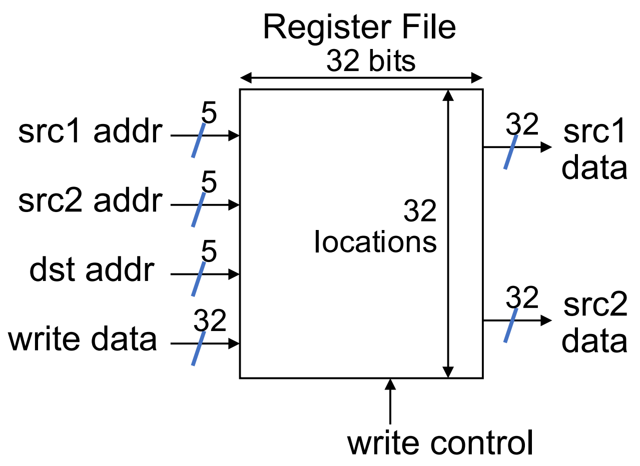
opcode	6-bits, opcode that specifies the operation
rs1	5-bits, register file address of the first source operand
rs2	5-bits, register file address of the second source operand
rd	5-bits, register file address of the result's destination
imm	12-bits / 20-bits, immediate number field
funct	3-bits / 10-bits, function code augmenting the opcode

The RISC-V ISA

Instruction Categories

- Load and Store instructions
- Bitwise instructions
- Arithmetic instructions
- Control transfer instructions
- Pseudo instructions

RISC-V Register File



- Holds thirty-two 32-bit general purpose registers
- Two read ports
- One write port

Registers are

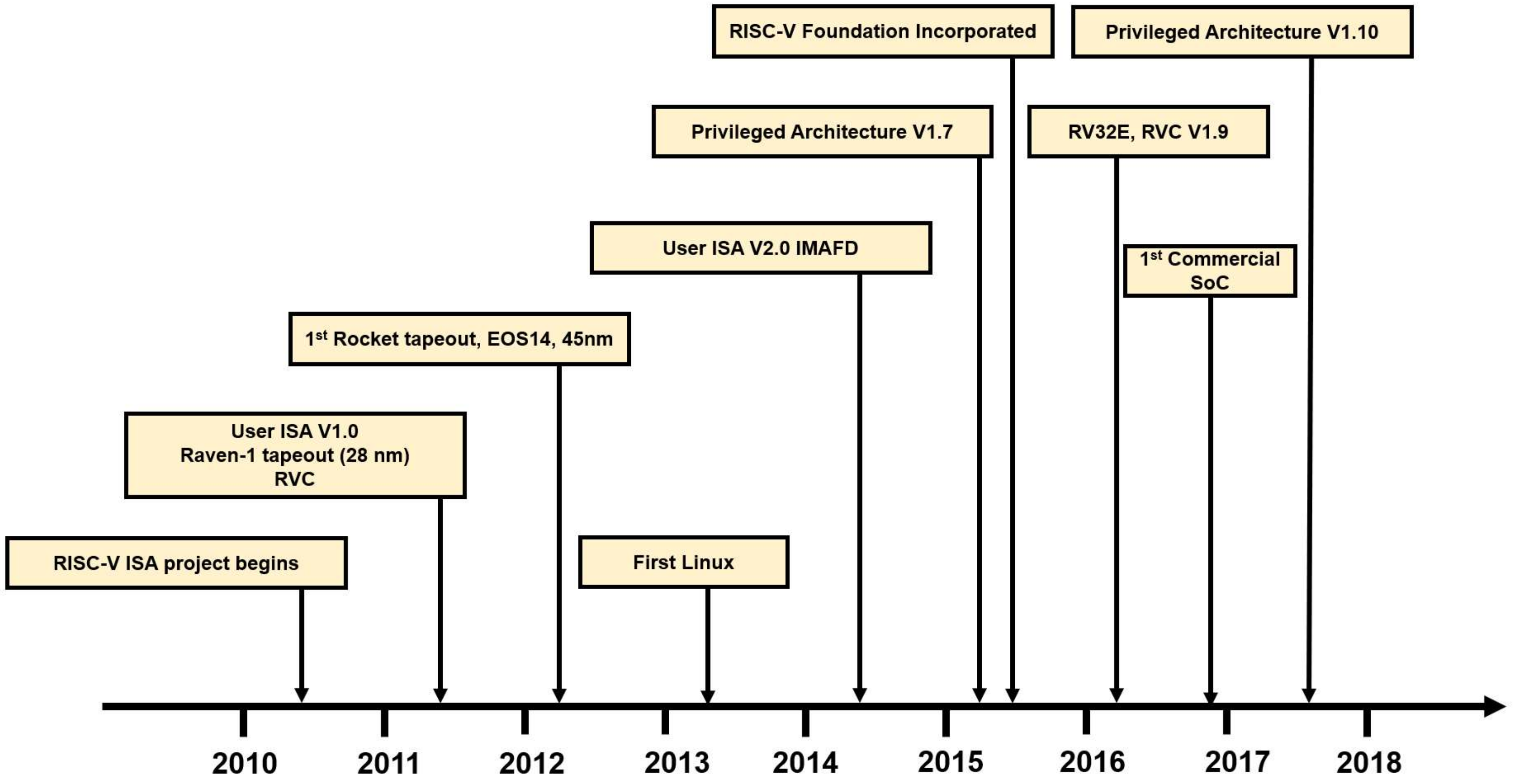
- Faster than main memory
  - But register files with more locations are slower
  - E.g., a 64 word file may be 50% slower than a 32 word file
  - Read/write port increase impacts speed quadratically
- Easier for a compiler to use
  - `(A*B)-(C*D)-(E*F)` can do multiplies in any order vs. stack
- Can hold variables so that code density improves (since register are named with fewer bits than a memory location)

Aside: RISC-V Register Convention

Table: Register names and descriptions

Register Names	ABI Names	Description
x0	zero	Hard-wired zero
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5	t0	Temporary / Alternate link register
x6-7	t1 - t2	Temporary register
x8	s0 / fp	Saved register / Frame pointer
x9	s1	Saved register
x10-11	a0-a1	Function argument / Return value registers
x12-17	a2-a7	Function argument registers
x18-27	s2-s11	Saved registers
x28-31	t3-t6	Temporary registers

History of RISC-V


RISC-V Foundation Incorporated
Privileged Architecture V1.10
Privileged Architecture V1.7
RV32E, RVC V1.9
User ISA V2.0 IMAFD
1st Commercial SoC
1st Rocket tapeout, EOS14, 45nm
User ISA V1.0
Raven-1 tapeout (28 nm)
RVC
RISC-V ISA project begins
First Linux
2010
2011
2012
2013
2014
2015
2016
2017
2018

Overview

1 Introduction

2 Control Instructions

3 Others

4 Summary

RISC-V Control Flow Instructions

RISC-V conditional branch instructions:

```
bne s0, s1, Lbl     # go to Lbl if s0 != s1
beq s0, s1, Lbl     # go to Lbl if s0 = s1
```

Example

Text

C: `if (i==j) h = i + j;`

```
      bne s0, s1, Lbl1
      add s3, s0, s1
Lbl1:  ...
```

(assume s0=i; s1=j; s3=h)

- Instruction Format (B format)   B-format: no imm[0]
- How is the branch destination address specified ?

Q: why we don't need imm[0]? note imm is the target address.
— imm[0] is always zero, because address is word aligned. (for 32-bit cpu, one word contains 4 bytes)

In Support of Branch Instructions

- We have `beq`, `bne`, but what about other kinds of branches (e.g., branch-if-less-than)?
- For this, we need yet another instruction, `slt` (less than set)

Set on less than instruction:

```
slt t0, s0, s1        # if s0 < s1   then
                      # t0 = 1       else
                      # t0 = 0
```

- Instruction format (R format or I format)

slti

Alternate versions of `slt`

```
slti  t0, s0, 25     # if s0 < 25 then t0 = 1 ...
sltu  t0, s0, s1     # if s0 < s1 then t0 = 1 ...
sltiu t0, s0, 25     # if s0 < 25 then t0 = 1 ...
```

sltu: slt instruction for unsigned values

Aside: More Branch Instructions

Can use `slt`, `beq`, `bne`, and the fixed value of 0 in register `zero` to create other conditions

- less than: `blt s1, s2, Label`

```
slt  t0, s1, s2         # t0 set to 1 if
bne  t0, zero, Label    # s1 < $s2
```

- less than or equal to: `ble s1, s2, Label`
- greater than: `bgt s1, s2, Label`
- great than or equal to: `bge s1, s2, Label`
- Such branches are included in the instruction set as pseudo instructions – recognized (and expanded) by the assembler

Bounds Check Shortcut

- Treating signed numbers as if they were unsigned gives a low cost way of checking if $0 \leq x < y$ (index out of bounds for arrays)

```
sltu t0, s1, t2          # t0 = 0 if
                         # s1 > t2  (max)   (we know t2 is larger than 0)
                         # or s1 < 0  (min)
beq  t0, zero, IOOB      # go to IOOB if
                         # t0 = 0
```

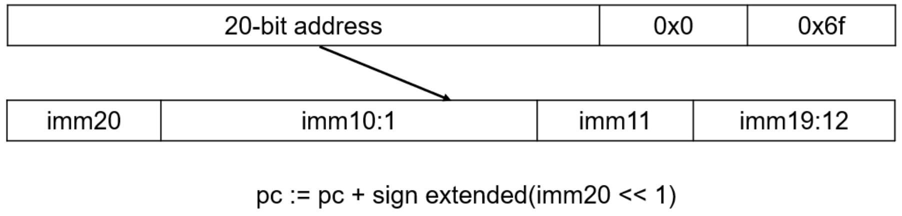
- The key is that negative integers in two's complement look like large numbers in unsigned notation.
- Thus, an unsigned comparison of x < y also checks if x is negative as well as if x is less than y.

Other Control Flow Instructions

- RISC-V also has an unconditional branch instruction or jump instruction:

```
jal zero, label          # go to label, label can be an
    immediate value
```

- Instruction Format (J Format)
- `J` is a pseudo instruction of unconditional `jal` and it will discard the return address (e.g., `j label`)



EX-2: Branching Far Away

What if the branch destination is further away than can be captured in 12 bits? Re-write the following codes.

```
beq     s0, s1, L1
```

Solution:

EX-2: Branching Far Away

What if the branch destination is further away than can be captured in 12 bits? Re-write the following codes.

```
        beq    s0, s1, L1
```

note: beq the imm range is small, so we can use J instruction, if target is further away.

Solution:

```
    bne s0, s1, L2
   j L1
L2:  ...
```

`j` is a pseudo instruction for jal (i.e., `jal zero, label`)

EX: Compiling a while Loop in C

```
    while (save[i] == k) i += 1;
```

Assume that `i` and `k` correspond to registers `s3` and `s5` and the base of the array save is in `s6`.

EX: Compiling a while Loop in C

```
while (save[i] == k) i += 1;
```

Assume that i and k correspond to registers s3 and s5 and the base of the array save is in s6.

(left shift 2 == x 4)

slli

```
Loop: sll  t1, s3, 2     # Temp reg t1 = i * 4
      add  t1, t1, s6    # t1 = address of save[i]
      lw   t0, 0(t1)     # Temp reg t0 = save[i]
      bne  t0, s5, Exit  # go to Exit if save[i] != k
      addi s3, s3,1      # i = i + 1
      j    Loop          # j is a pseudo instruction for jal
                         # go to Loop
Exit:
```

t1: address shift in array

Note: left shift s3 to align word address, and later address is increased by 1

32-bit CPU: one word == 32bits == 4 bytes
64-bit CPU: one word == 64bits == 8 bytes

Six Steps in Execution of a Procedure

1. Main routine (caller) places parameters in a place where the procedure (callee) can access them
   - `a0` – `a7`: four argument registers
2. Caller transfers control to the callee
3. Callee acquires the storage resources needed
4. Callee performs the desired task
5. Callee places the result value in a place where the caller can access it
   - `s0–s11`: 12 value registers for result values
6. Callee returns control to the caller
   - `ra`: one return address register to return to the point of origin

Instructions for Accessing Procedures

add xxx
calling address: → jal ra, label
return address: → …
…

label:

We have learnt `jal`, now let's continue

- RISC-V procedure call instruction:

```
jal  ra, label  # jump and link,
                # label can be an immediate value
```

- Saves PC + 4 in register `ra` to have a link to the next instruction for the procedure return

PC: point counter. current instruction location.
when we move to next instruction, PC+4

- Machine format (J format):
- Then can do procedure return with a

```
jalr x0, 0(ra)  # return
```

- Instruction format (I format)

Example of Accessing Procedures

- For a procedure that computes the GCD of two values i (in `t0`) and j (in `t1`): `gcd(i,j);`
- The caller puts the i and j (the parameters values) in `a0` and `a1` and issues a

```
    jal ra, gcd        # jump to routine gcd
```

- The callee computes the GCD, puts the result in `s0`, and returns control to the caller using

```
gcd: . . .          # code to compute gcd
     jalr x0, 0(ra)         # return
```

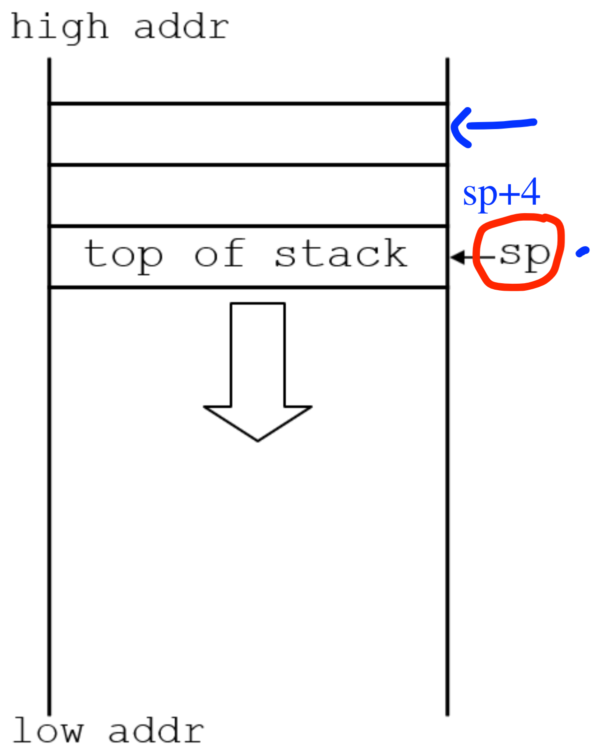
What if the callee needs to use more registers than allocated to argument and return values?

- Use a stack: a last-in-first-out queue
- One of the general registers, `sp`, is used to address the stack
- "grows" from high address to low address
- push: add data onto the stack, data on stack at new `sp`

```
sp = sp - 4
```

- pop: remove data from the stack, data from stack at `sp`

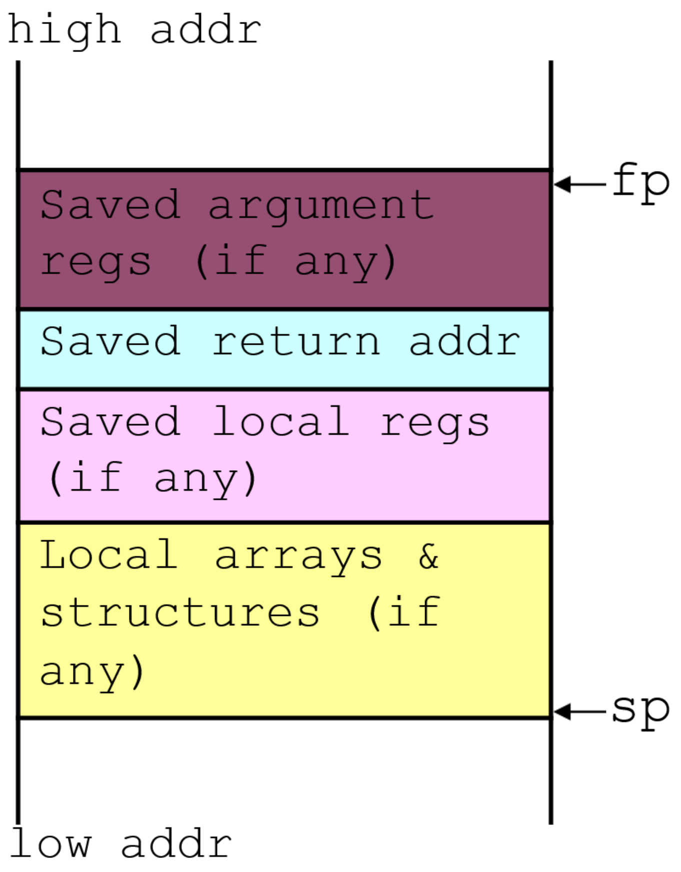
```
sp = sp + 4
```



Allocating Space on the Stack

- The segment of the stack containing a procedure's saved registers and local variables is its procedure frame (aka activation record)
- The frame pointer (`fp`) points to the first word of the frame of a procedure – providing a stable "base" register for the procedure
- `fp` is initialized using `sp` on a call and `sp` is restored using `fp` on a return

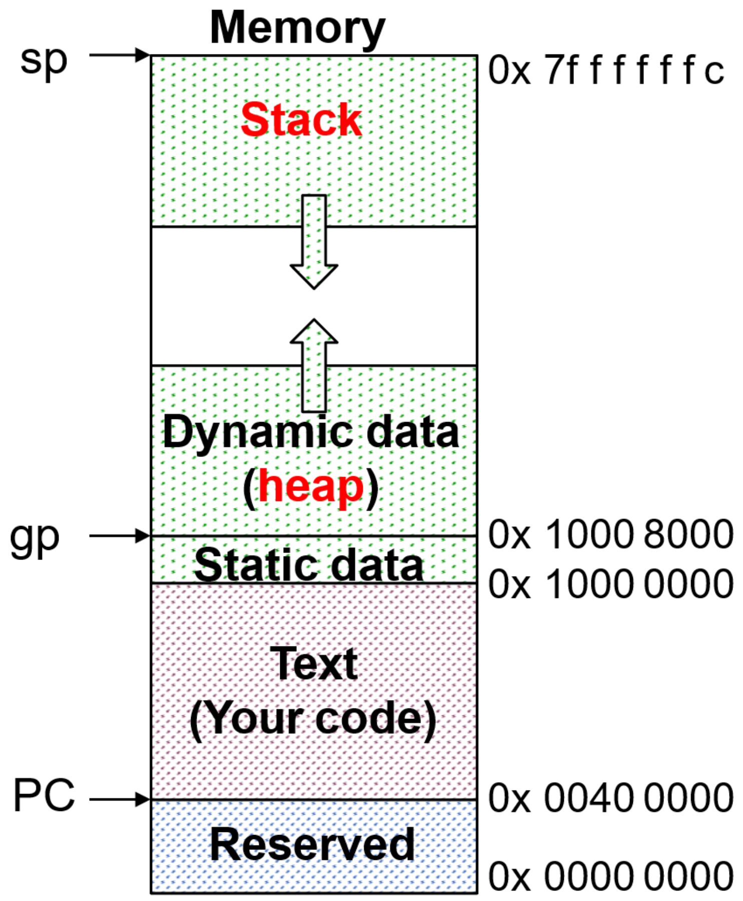
fp: sp location when we starts the procedure



high addr
fp
Saved argument regs (if any)
Saved return addr
Saved local regs (if any)
Local arrays & structures (if any)
sp
low addr

Allocating Space on the Heap

- Static data segment for constants and other static variables (e.g., arrays)
- Dynamic data segment (aka heap) for structures that grow and shrink (e.g., linked lists)
- Allocate space on the heap with `malloc()` and free it with `free()` in C



EX-3: Compiling a C Leaf Procedure

Leaf procedures are ones that do not call other procedures. Give the RISC-V assembler code for the follows.

```
int leaf_ex (int g, int h, int i, int j)
{
    int f;
    f = (g+h) - (i+j);
    return f;
}
```

Solution:

EX-3: Compiling a C Leaf Procedure

Leaf procedures are ones that do not call other procedures. Give the RISC-V assembler code for the follows.

```
int leaf_ex (int g, int h, int i, int j)
{
    int f;
    f = (g+h) - (i+j);
    return f;
}
```

Solution:

Suppose g, h, i, and j are in `a0, a1, a2, a3`

```
leaf_ex:  addi   sp, sp, -8  # make stack room
          sw     t1, 4(sp)   # save t1 on stack
          sw     t0, 0(sp)   # save t0 on stack
          add    t0, a0, a1
          add    t1, a2, a3
          sub    s0, t0, t1
          lw     t0, 0(sp)   # restore t0
          lw     t1, 4(sp)   # restore t1
          addi   sp, sp, 8   # adjust stack ptr
          jalr   zero, 0(ra)
```

(we push two values into the stack)

Q: why we need to store t0 & t1 into the stack?
—other procedures may also use t0 & t1;
since locally we may use t0 & t1, we need to backup their original values

Return to caller

contain address of caller